– Independent Work Report Fall, 2015 –

Verification of a Deterministic Random Bits Generator

Naphat Sanguansin
Adviser: Andrew W. Appel

Abstract

In this project, I formally verified the ***mbedTLS*** *implementation of a HMAC-based deterministic random bits generator (DRBG) with respect to the standard* ***NIST SP 800-90A****. By "formal", I mean the verification is in the machine-checked environment of Coq using the Verified Software Toolchain (VST) framework. I verified the functions* `update`*,* `reseed`*, and* `generate`*. Due to time constraint, I was not able to verify the* `instantiate` *function.*

1. Introduction

A deterministic random bits generator (DRBG) is a program that takes a source of randomness, or entropy, and stretches it out into a much longer string of pseudorandom bits. **NIST SP 800-90A** [2] DRBG provides backtracking resistance and, optionally, prediction resistance. Backtracking resistance means that even if the whole (secret) state of the DRBG is known at time t, an adversary will not be able to distinguish between a truly random bitstring and a bitstring generated by the DRBG at some time before t. Prediction resistance is the complement, providing the same assurance for some time including and after t. Here, "time" is counted in terms of calls to the `generate` function.

NIST SP 800-90A specifies different types of DRBG. The version I verified in this project is the one based on the key-hashed message authentication code, or HMAC. HMAC takes a (secret) key and a message and computes a fixed-length message authentication code. In theory, someone with only the message will not be able to compute the message authentication code, and someone with both the message and the message authentication code will not be able to compute the key. Note that HMAC is an irreversible function.

Why did I choose to verify an implementation of DRBG? In short, DRBG is a widely used crypto primitive. Correctly implemented crypto primitives help application programmers improve the security of their code without them having to fully understand the math behind the crypto algorithms.

1.1. Machine-Checked Proofs

I did the verification in Coq, an open-sourced proof assistant. The C code is compiled into its abstract syntax tree (AST) in Coq using the CompCert certified (proved correct with a machine-checked proof) compiler. I wrote functional specifications (functional programs) in Coq that closely match **NIST SP 800-90A**. Using the functional programs, I created API specifications using the VST framework. API specifications reason about how the AST deals with memory and local variables. I then proved the AST correct with respect to the API specifications.

Of what the tasks I mentioned, the ones that must be trusted are the translation of **NIST SP 800-90A** into functional specifications and the creation of API specifications from those functional programs. Everything else is machine-checked. Can the functional and API specifications be trusted? For the latter, I hope to make a convincing claim by the end of this paper that as long as the functional specifications are correct, the corresponding API specifications will be, too. As for the functional specifications, first of all, they are small, easy-to-reason-about, strongly typed functional programs. I also ran the programs on some test inputs from NIST and verified that the output matches. That said, it is still unsatisfying that at the functional specification level, I had to resort to unit tests. Thankfully, there is a parallel project by Katherine Ye ‘16 that will prove cryptographic properties of the functional specifications. The fact that this parallel project can be done concurrent to (and mostly independent from) the verification of the C implementation shows the power and flexibility of VST.

1.2. Related Works

Similar verifications have been done for SHA-256 and HMAC, see [1] and [3] respectively. As mentioned previously, Katherine Ye ‘16 is working on proving cryptographic properties of the functional specifications.

As far as I could tell, this is the first end-to-end verification of an implementation of HMAC DRBG against a functional specification. By “end-to-end” I mean that the verification builds upon an existing verification of HMAC [3] (which is, in turn, built upon an existing verification of SHA-256 [1]).

There have been some other verification efforts to verify a DRBG, albeit without using a functional specification. As an example, Dörre and Klebanov [4] verified Android’s pseudorandom number generator. This verification is done on a Java implementation, and is a verification on information flow, not on a functional specification. That is, the project proves that all the bytes of entropy are used in constructing the secret internal state, but not *how* the bytes are used. Proving an implementation against a functional specification is arguably better, as it allows the functional specification to serve as a mathematical representation of the more complex C (or Java) implementation. This functional specification can then be shown to have desired cryptographic properties, including the fact that all bytes of entropy are being properly consumed.

1.3. Why mbedTLS?

Previous efforts in the VST project to verify SHA-256 [1] and HMAC [3] have done so using **OpenSSL 0.9.1c**. A natural choice for the DRBG verification would be to also use the same **OpenSSL** library. Unfortunately, the DRBG implementation in **OpenSSL** makes heavy use of function pointers to encapsulate the differences between each type of DRBG. While Verifiable C, the C program logic component of the VST framework, is capable of reasoning about function pointers, the proof automation system is not yet robust enough when dealing with function pointers and so such proofs will be more complex. **mbedTLS**, on the other hand, does not make use of function pointers except to deal with entropy. Therefore, I chose to use the implementation of DRBG found in **mbedTLS 2.1.1**.

1.4. Modifications to mbedTLS

I had to make a few modifications to the **mbedTLS** DRBG source file. Most of these transformations are obviously equivalent to the original version. For example, Verifiable C does not allow memory dereferences in function parameters or if/while/for-loop conditions. Therefore, code such as the following had to be changed:

```
md_len = mbedtls_md_get_size( ctx→md_ctx.md_info );
```

to this:

```
info = ctx→md_ctx.md_info;
md_len = mbedtls_md_get_size( info );
```

The other modification I had to make was to eliminate the use of function pointers when dealing with entropy.

2. A Functional Representation of Entropy

A DRBG needs some entropy input to act as a source of randomness. According to **NIST SP 800-90A**, the source of entropy input can be either an approved entropy source, a nondeterministic random bits generator (NRBG), or another DRBG. The entropy API used in NIST is the following:

```
(status, entropy_input) = Get_entropy_input(min_entropy, min_length,
    max_length, prediction_resistance_request)
```

`min_entropy` refers to how many bits of entropy to use. `min_length` and `max_length` bounds the length of the output string of bits. If `prediction_resistance_request` is set, then fresh bits of entropy are used.

For this project, I wanted a simple entropy representation that still satisfies **NIST SP 800-90A**. A flawed idea is to represent entropy as an infinite stream of `bool`, each `bool` representing a bit. This does not model real-life entropy. Entropy is expensive; if it was not, there would be no need for a DRBG to begin with! Requesting too many bits of entropy when there is insufficient supply can cause a failure. To capture this behavior, I chose to model entropy as an infinite stream of `option bool`.

In Coq, as well as many typed programming languages, an `option` type is a type whose members can either exist and have a value or not exist. Types in C and Java are implicitly optional, with the members that do not exist being represented by `null`. It is effectively equivalent to this:

```
Inductive option (T: Type) :=
| Some: T →option T
| None: option T.
```

A `None` value means that the entropy bit is not yet available, and so the call to get entropy should fail. An infinite stream of entropy can be defined as follows:

```
Definition stream: Type := nat →option bool.
```

The `nat` (natural number) argument is used to index into the stream. Note that a simple `list` would not work here, because we need the stream to have infinite size.

Any functions that consume entropy will need to return a new entropy stream. Furthermore, because calls to get entropy can fail, such functions have to be able to handle failures. For this reason, I defined a general `result` datatype:

```
Inductive error_code: Type :=
| catastrophic_error
| generic_error.


Inductive result (X: Type): Type :=
| success: X →stream →result X
| error : error_code →stream →result X.
```

The name `catastrophic_error` came from **NIST SP 800-90A** and is used to denote errors that are not user errors. An example of user errors is requesting prediction resistance when the

DRBG state was not initialized with prediction resistance capability. An example of catastrophic errors is the entropy source failing to provide the requested amount of entropy.

I also defined a `get_bits` function that, if possible, gets the consecutive requested number of bits from the stream and returns a new stream that advances the index by the requested number of bits. If `get_bits` encounters an error (a `None` value), it returns a new stream that skips this error the next time it is called, symbolizing a movement forward in time resulting in more entropy being available.

This stream representation of entropy is a specialized case of the `Get_entropy_input` function from **NIST SP 800-90A**, with each bit of the stream representing a bit of entropy, therefore setting `min_entropy = min_length = max_length`, with prediction resistance always on. In the language of **NIST SP 800-90A**, this entropy representation is an "ideal random bitstring".

3. DRBG Algorithm

The DRBG algorithm contains four public functions: `instantiate`, `reseed`, `generate`, and `uninstantiate`. The HMAC-based DRBG also makes use of a helper function: `update`. I will focus on all but `uninstantiate`, as it simply involves freeing memory.

3.1. The DRBG State

NIST SP 800-90A §10.1.2.1 describes the HMAC DRBG state. In the functional specification, I chose to represent the HMAC DRBG state as the following, which closely resembles the NIST specifications.

```
Definition DRBG_working_state :=
  (list Z * list Z * Z). (* value * key * reseed_counter *)
Definition DRBG_state_handle :=
  (DRBG_working_state * Z * bool). (* state, security_strength,
     prediction_resistance_flag *)
```

The two most important fields here are `key` and `value`. The `key` is the secret internal state, used in conjunction with HMAC, while the `value` is the “output” of the DRBG. I will explain what “output” means when I explain the `generate` function (§3.5).

The distinction between a “working state” and a “state handle” is that **NIST SP 800-90A** presents each of the functions as first doing error checking then calling an inner algorithm. The error checking part is common to all types of DRBG, not just the HMAC-based one. The inner algorithm is specific to each type of DRBG, and the algorithm only needs the information stored in the working state (and the working state type does not need to be the same for each type of DRBG). See **NIST SP 800-90A §8.3** for more details.

`Z` is Coq’s representation of arbitrary-precision integers, and is used here to represent a byte. That is, the value of the integer must be between 0 and 255, inclusive. While I could have represented each byte as a tuple of a `Z` and a proof that the value of that `Z` is in the byte range, keeping that fact separate allowed me to write a functional specification that closely resembles **NIST SP 800-90A**. The fact that if the input state contains bytes, then the output will also contain bytes can easily be provided as separate lemmas about the individual functions.

3.2. The Update Function

Before showing any of the public functions, I must first define a helper function, `update`. See **NIST SP 800-90A §10.1.2.2** for the corresponding NIST specification. The `update` function takes a `key`, a `value`, and an optional nonce and returns a new `key` and a new `value`. This function will be called at least once in each of the public functions, and the goal is to provide backtracking resistance. This is achieved by having the secret internal state of the DRBG (the `key`) be the output of HMAC, an irreversible function.

```
Definition HMAC_DRBG_update (HMAC: list Z →list Z →list Z)
    (provided_data K V: list Z): (list Z * list Z) :=
  let K := HMAC (V ++ [0] ++ provided_data) K in
  let V := HMAC V K in
  match provided_data with
```

```
    | [] ⇒ (K, V)
    | _::_ ⇒
      let K := HMAC (V ++ [1] ++ provided_data) K in
      let V := HMAC V K in
      (K, V)
  end.
```

Here, `provided_data` is the aforementioned optional nonce.

3.3. The Instantiate Function

The `instantiate` function initializes the DRBG state with randomness from a source of entropy. The caller is responsible for specifying the minimum security strength for this DRBG, which for an HMAC-based DRBG translates to "how many bits of entropy should be requested?". In addition to entropy, the DRBG also asks for a nonce, which is roughly defined as a string of bits with at least `1/2 security_strength` bits of entropy. In fact, this nonce can even be taken from the same source of entropy. In the functional specification, I am supporting both behaviors by having a parameter `provided_nonce: option (list Z)` that is used as the nonce if a value is given. In the case that `provided_nonce` is `None`, the nonce is taken from the entropy source.

Here is the functional specification I wrote for the `instantiate` function (See **NIST SP 800-90A §9.1** for the corresponding NIST specification):

```
Definition DRBG_instantiate_function
  (instantiate_algorithm: list Z →list Z →list Z →Z →
     DRBG_working_state)
  (min_entropy_length max_entropy_length: Z)
  (provided_nonce: option (list Z))
  (highest_supported_security_strength: Z)
  (max_personalization_string_length: Z)
  (prediction_resistance_supported: bool)
```

```
(entropy_stream: ENTROPY.stream)
(requested_instantiation_security_strength: Z)
(prediction_resistance_flag: bool)
(personalization_string: list Z)
: ENTROPY.result DRBG_state_handle :=
 if requested_instantiation_security_strength >?
    highest_supported_security_strength
  then ENTROPY.error ENTROPY.generic_error entropy_stream
 else match prediction_resistance_flag,
    prediction_resistance_supported with
      | true, false ⇒ ENTROPY.error ENTROPY.generic_error
        entropy_stream
      | _,_ ⇒
       if (Zlength personalization_string) >?
         max_personalization_string_length then ENTROPY.error
         ENTROPY.generic_error entropy_stream
       else
        let security_strength :=
         if requested_instantiation_security_strength <=? 14 then
           Some 14
         else if requested_instantiation_security_strength <=? 16
           then Some 16
         else if requested_instantiation_security_strength <=? 24
           then Some 24
         else if requested_instantiation_security_strength <=? 32
           then Some 32
         else None in
```

```
match security_strength with
  | None ⇒ ENTROPY.error ENTROPY.generic_error
     entropy_stream
  | Some security_strength ⇒
  match get_entropy security_strength min_entropy_length
     max_entropy_length prediction_resistance_flag
     entropy_stream with
    | ENTROPY.error e s' ⇒ ENTROPY.error
       ENTROPY.catastrophic_error s'
    | ENTROPY.success entropy_input entropy_stream ⇒
     let nonce_result :=
      match provided_nonce with
        | Some n ⇒ ENTROPY.success n entropy_stream
        | None ⇒ get_entropy (security_strength/2)
           (min_entropy_length/2) (max_entropy_length/2)
           prediction_resistance_flag entropy_stream
      end in
     match nonce_result with
      | ENTROPY.error e s' ⇒ ENTROPY.error
         ENTROPY.catastrophic_error s'
      | ENTROPY.success nonce entropy_stream ⇒
       let initial_working_state := instantiate_algorithm
           entropy_input nonce personalization_string
           security_strength in
       ENTROPY.success (initial_working_state,
           security_strength, prediction_resistance_flag)
           entropy_stream
```

```
                    end
                  end
                end
             end.
```

This is just the "outer" function that does error checking on the parameters, as mentioned earlier. This outer function is purely generic; different types of DRBG can share this piece of code. The actual algorithm for computing an initial working state is left as a dependency in the first parameter. All parameters before `entropy_stream` are implementation-specific constants. For example, `prediction_resistance_supported` is a flag denoting whether or not this particular implementation can provide prediction resistance.

Of the actual parameters of this function (everything from and including `entropy_stream`), `security_strength` has already been explained. `prediction_resistance_flag` is used to indicate whether this particular instantiation of DRBG will provide prediction resistance. `personalization_string` is expected to be unique, and can include things like the device identifier, timestamp, etc. The idea is not to provide more randomness but to ensure that this particular instantiation will be different from any other instantiations.

The HMAC-based algorithm for instantiation is as follows (See **NIST SP 800-90A §10.1.2.3** for the corresponding NIST specification):

```
Definition HMAC_DRBG_instantiate_algorithm (HMAC: list Z →list Z →
    list Z) (entropy_input nonce personalization_string: list Z)
    (security_strength: Z): DRBG_working_state :=
  let seed_material := entropy_input ++ nonce ++ personalization_string
     in
  let key := initial_key in
  let value := initial_value in
  let (key, value) := HMAC_DRBG_update HMAC seed_material key value in
  let reseed_counter := 1 in
  (value, key, reseed_counter).
```

`initial_key` and `initial_value` are globally defined constants of length equal to the length of the HMAC output. `initial_key` is a string of all byte 0, while `initial_value` is a string of all byte 1.

I chose to leave the actual HMAC function as a parameter, to allow different implementations to be swapped in (the NIST specification works for all approved HMAC functions).

3.4. The Reseed Function

There are times when the DRBG state must be reseeded with new bits of entropy. This can happen because `generate` has been called many times with the current bits of entropy, or because prediction resistance is requested. Reseeding with fresh entropy provides prediction resistance as the new secret state is a (irreversible) function of the current secret state and some random, unknown entropy. Note that reseeding is preferred over simply instantiating a new DRBG state because if the source of entropy fails silently (supposed the bits of entropy returned are not actually random), at least the DRBG still has some randomness left over from the old entropy.

The functional specification for the "outer" reseed function follows. Again, remember that this is the part that does error checking and is common among all types of DRBG. See **NIST SP 800-90A §9.2** for the corresponding NIST specification.

```
Definition DRBG_reseed_function
  (reseed_algorithm: DRBG_working_state →list Z →list Z →
     DRBG_working_state)
  (min_entropy_length max_entropy_length: Z)
  (max_additional_input_length: Z)
  (entropy_stream: ENTROPY.stream)
  (state_handle: DRBG_state_handle)
  (prediction_resistance_request: bool)
  (additional_input: list Z)
```

```
: ENTROPY.result DRBG_state_handle :=
match state_handle with (working_state, security_strength,
  prediction_resistance_flag) ⇒
if prediction_resistance_request && (negb prediction_resistance_flag)
 then ENTROPY.error ENTROPY.generic_error entropy_stream
else
 if (Zlength additional_input) >? max_additional_input_length
  then ENTROPY.error ENTROPY.generic_error entropy_stream
 else
  match get_entropy security_strength min_entropy_length
      max_entropy_length prediction_resistance_flag entropy_stream
      with
    | ENTROPY.error _s ⇒ ENTROPY.error ENTROPY.catastrophic_error s
    | ENTROPY.success entropy_input entropy_stream ⇒
      let new_working_state := reseed_algorithm working_state
          entropy_input additional_input in
      ENTROPY.success (new_working_state, security_strength,
        prediction_resistance_flag) entropy_stream
  end
end.
```

As with `instantiate`, the parameters up to but excluding `entropy_stream` are implementation-specific constants.

Both the `reseed` and `generate` functions allow users to pass in an additional sequence of bytes. Users do not need to provide this (and implementations can choose to omit this feature), but it is meant to provide any additional bits of entropy or to provide personalization of this specific call.

The HMAC-based reseed algorithm follows (See **NIST SP 800-90A §10.1.2.4** for the corresponding NIST specification):

```
Definition HMAC_DRBG_reseed_algorithm (HMAC: list Z →list Z →list Z)
    (working_state: DRBG_working_state) (entropy_input
    additional_input: list Z): DRBG_working_state :=
  match working_state with (v, key, _) ⇒
    let seed_material := entropy_input ++ additional_input in
    let (key, v) := HMAC_DRBG_update HMAC seed_material key v in
    let reseed_counter := 1 in
    (v, key, reseed_counter)
  end.
```

3.5. The Generate Function

The `generate` function is used to generate an arbitrary number of pseudorandom bytes, automatically calling `reseed` first if necessary. A reseed is necessary if the caller asks for prediction resistance or if the inner (HMAC-based) `generate` algorithm returns a flag saying a reseed is necessary. Note that the DRBG state has to have been initialized with prediction resistance capability if the caller wants prediction resistance.

The outer function body for `generate` follows. See **NIST SP 800-90A §9.3** for the corresponding NIST specification. Unlike with the previous functions, a recursive helper function is needed to specify this function body because of the use of a `goto` statement in **NIST SP 800-90A**. The recursive helper function captures the "reseed if required" logic.

```
Fixpoint DRBG_generate_function_helper
  (generate_algorithm: DRBG_working_state →Z →list Z →
     DRBG_generate_algorithm_result)
  (reseed_function: ENTROPY.stream →DRBG_state_handle →bool →list Z
    →ENTROPY.result DRBG_state_handle)
  (entropy_stream: ENTROPY.stream)
  (state_handle: DRBG_state_handle)
```

```
(requested_number_of_bytes: Z)
(prediction_resistance_request: bool)
(additional_input: list Z)
(should_reseed: bool)
(count: nat)
: ENTROPY.result (list Z * DRBG_working_state) :=
let result := if should_reseed then
              match reseed_function entropy_stream state_handle
                  prediction_resistance_request additional_input with
                | ENTROPY.success x entropy_stream ⇒ ENTROPY.success
                    (x, []) entropy_stream
                | ENTROPY.error e entropy_stream ⇒ ENTROPY.error e
                    entropy_stream
              end
            else ENTROPY.success (state_handle, additional_input)
                entropy_stream in
match result with
  | ENTROPY.error e s ⇒ ENTROPY.error e s
  | ENTROPY.success (state_handle, additional_input) entropy_stream
      ⇒
    match state_handle with (working_state, security_strength,
        prediction_resistance_flag) ⇒
      match generate_algorithm working_state requested_number_of_bytes
          additional_input with
        | generate_algorithm_reseed_required ⇒
          match count with
            | O ⇒ ENTROPY.error ENTROPY.generic_error entropy_stream (*
            impossible *)
          | S count' ⇒ DRBG_generate_function_helper
            generate_algorithm reseed_function entropy_stream
            state_handle requested_number_of_bytes
            prediction_resistance_request additional_input true
            count'
        end
      | generate_algorithm_success x y ⇒ ENTROPY.success (x, y)
          entropy_stream
    end
  end
end.


Definition DRBG_generate_function
  (generate_algorithm: Z →DRBG_working_state →Z →list Z →
     DRBG_generate_algorithm_result)
  (reseed_function: ENTROPY.stream →DRBG_state_handle →bool →list Z
   →ENTROPY.result DRBG_state_handle)
  (reseed_interval: Z)
  (max_number_of_bytes_per_request: Z)
  (max_additional_input_length: Z)
  (entropy_stream: ENTROPY.stream)
  (state_handle: DRBG_state_handle)
  (requested_number_of_bytes requested_security_strength: Z)
  (prediction_resistance_request: bool)
  (additional_input: list Z)
  : ENTROPY.result (list Z * DRBG_state_handle) :=
match state_handle with (working_state, security_strength,
    prediction_resistance_flag) ⇒
  if requested_number_of_bytes >? max_number_of_bytes_per_request
    then ENTROPY.error ENTROPY.generic_error entropy_stream
  else
    if requested_security_strength >? security_strength
      then ENTROPY.error ENTROPY.generic_error entropy_stream
    else
      if (Zlength additional_input) >? max_additional_input_length
        then ENTROPY.error ENTROPY.generic_error entropy_stream
      else
        if prediction_resistance_request && (negb
            prediction_resistance_flag)
          then ENTROPY.error ENTROPY.generic_error entropy_stream
        else
          match DRBG_generate_function_helper (generate_algorithm
              reseed_interval) reseed_function entropy_stream
              state_handle requested_number_of_bytes
              prediction_resistance_request additional_input
              prediction_resistance_request 1%nat with
            | ENTROPY.error e s ⇒ ENTROPY.error e s
            | ENTROPY.success (output, new_working_state) entropy_stream
                ⇒
              ENTROPY.success (output, (new_working_state,
                  security_strength, prediction_resistance_flag))
                  entropy_stream
          end
```

```
end.
```

A **Fixpoint** is a recursive function where at least one parameter is structurally getting smaller, allowing Coq to figure out the proof of termination automatically. In this case, it is the last parameter, the natural number `count`, which is initialized with `1`, meaning that this recursive function is only allowed to go one level deep.

As usual, the `generate` algorithm is left as a dependency. Also left as a dependency is the `reseed` function. The signature matches the `reseed` function shown earlier, albeit with the dependencies and implementation constants filled in.

The HMAC-based `generate` algorithm follows. See **NIST SP 800-90A §10.1.2.5** for the corresponding NIST specification. A recursive function is also needed to specify the algorithm, due to the use of a `while` loop in **NIST SP 800-90A**.

```
Function HMAC_DRBG_generate_helper_Z (HMAC: list Z →list Z →list Z)
    (key v: list Z) (requested_number_of_bytes: Z) {measure Z.to_nat
    requested_number_of_bytes}: (list Z * list Z) :=
    if 0 ≥? requested_number_of_bytes then (v, [])
    else
      let len := 32%nat in
      let (v, rest) := HMAC_DRBG_generate_helper_Z HMAC key v
        (requested_number_of_bytes -(Z.of_nat len)) in
      let v := HMAC v key in
      let temp := v in
      (v, rest ++ temp).


Definition HMAC_DRBG_generate_algorithm (HMAC: list Z →list Z →list
    Z) (reseed_interval: Z) (working_state: DRBG_working_state)
    (requested_number_of_bytes: Z) (additional_input: list Z):
    DRBG_generate_algorithm_result :=
match working_state with (v, key, reseed_counter) ⇒
  if reseed_counter >? reseed_interval then
     generate_algorithm_reseed_required
  else
    let (key, v) := match additional_input with
                     | [] ⇒ (key, v)
                     | _::_ ⇒ HMAC_DRBG_update HMAC additional_input
                        key v
                    end in
    let (v, temp) := HMAC_DRBG_generate_helper_Z HMAC key v
       requested_number_of_bytes in
    let returned_bits := firstn (Z.to_nat requested_number_of_bytes)
       temp in
    let (key, v) := HMAC_DRBG_update HMAC additional_input key v in
    let reseed_counter := reseed_counter + 1 in
    generate_algorithm_success returned_bits (v, key, reseed_counter)
end.
```

A `Function` is the general form of a recursive function, and unlike `Fixpoint`, typically requires a proof that the function does terminate. In this case, the argument `requested_number_of_bytes` is decreasing, but in a nonobvious way (not structurally). The proof is simple, but long (about as long as the function body). Luckily, it is machine-checked, so I will not present it here.

The `generate` algorithm first checks if the number of times `generate` has been called (`reseed_counter`) is too high, and if so, returns a flag saying a reseed is needed. Otherwise, it repeatedly HMACs the current `value` with the current `key`, appends the result to the output, and stores the result as the new `value`. This is why the `value` part of the DRBG state is not considered secret, and is really the output of the DRBG. The output is then trimmed down to the requested number of bytes, and `reseed_counter` is incremented by `1`.

4. Verification

4.1. HMAC Interface

In this section, I explain the (pre-existing) interface of HMAC, of which the DRBG specification is a client.

The HMAC context is represented as a state machine with three states: `EMPTY`, `FULL key`, and `REP key data`. `EMPTY` refers to an uninitialized context. `FULL key` means the context has been used to digest some data with the key `key`, but it must be reset before it can be used again. That is, the difference between `EMPTY` and `FULL` is that `FULL` remembers the key the context was previously associated with. Lastly, `REP key data` means that the context is in the middle of digesting bytes `data` with the key `key`.

There are four functions associated with HMAC: `reset`, `start`, `update`, and `final`. `reset` takes a HMAC context in `FULL key` and turns it into `REP key []`, where `[]` is the empty byte sequence. `start` takes a HMAC context `EMPTY` and an additional parameter `key` and turns it into `REP key []`. `update` takes a HMAC context in `REP key data` and an additional parameter `data1` and turns it into `REP key (data ++ data1)`, where `++` is the list concatenation. Lastly, `final` takes a HMAC context in `REP key data` and a pointer to a space in memory `out` and turns the context into `FULL key` and deposits the output of `HMAC data key` into `out`.

The state machine representation of HMAC was created for this project. Beringer *et al.* [3] describe the previous verification effort on HMAC, but they represented HMAC in a more low-level format. See §5.1 for more details.

Even though I am verifying **mbedTLS** implementation of DRBG, the pre-existing verifications and specifications of SHA-256 and HMAC are for the **OpenSSL** version. To make the two versions work with each other, I created a thin wrapper on the **mbedTLS** HMAC interface. For example, take the `final` function. **mbedTLS** and **OpenSSL** have similar APIs for the `final` function, and I connect them with this wrapper function:

```
int mbedtls_md_hmac_finish( mbedtls_md_context_t *ctx, unsigned char
    *output) {
    HMAC_Final(ctx->hmac_ctx, output);
    return 0;
}
```

Here, `HMAC_Final` is the name of the `final` function verified by previous efforts, and `mbedtls_md_hmac_finish` is the name mbedTLS expects. The API specification (how the function interacts with local variable and memory) for this function is almost a complete copy of the specification defined by the previous HMAC verification efforts. The proofs that the wrapper functions satisfies the API specifications is also simple.

For more details on how mbedTLS and OpenSSL APIs are bridged, see Appendix A.

4.2. The *Functional*, *Abstract*, and *Concrete* DRBG States

Recall that in the functional specification (§3.1), I chose to represent the DRBG state as the following:

```
Definition DRBG_working_state :=
  (list Z * list Z * Z). (* value * key * reseed_counter *)
Definition DRBG_state_handle :=
  (DRBG_working_state * Z * bool). (* state, security_strength,
     prediction_resistance_flag *)
```

From here onwards, I will refer to this representation as the *functional* DRBG state.

When writing the API specification, I chose a representation that more closely resembles what is in **mbedTLS**. Naturally, **mbedTLS** represents the DRBG state as a C struct. Note that the original **mbedTLS** code includes a function pointer for the entropy function as well as context to pass to the entropy function. As noted earlier, for the purposes of this project, I made the modifications to remove this function pointer.

```
typedef struct mbedtls_hmac_drbg_context
{
   mbedtls_md_context_t md_ctx;
   unsigned char V[32];
   int reseed_counter;
   size_t entropy_len;
   int prediction_resistance;
   int reseed_interval;
} mbedtls_hmac_drbg_context;
```

The `key` in the functional DRBG state is implicitly represented in the HMAC context, `md_ctx`. Notice a couple differences from the functional DRBG state.

1. The field `entropy_len` in the C struct is equivalent to `security_strength` defined in the functional state.
2. The field `reseed_interval` is stored explicitly in the C struct but is treated as a global constant in NIST SP 800-90A and so in the functional specification.

There are two Coq representations for this C struct. The first, which I will call the *abstract* DRBG state, represents the mathematical equivalent of the C struct.

```
Inductive hmac256drbgabs :=
  HMAC256DRBGabs: ∀(key: list Z) (V: list Z) (reseed_counter
     entropy_len: Z) (prediction_resistance: bool) (reseed_interval:
     Z), hmac256drbgabs.
```

What I meant by "mathematical equivalent" is perhaps best demonstrated in the field `prediction_resistance`. In C, `prediction_resistance` is an `int`, meaning it can take any value between -2^{31} and $2^{31}-1$. However, logically, `prediction_resistance` is a boolean value, and there are two possible values: true and false. This is represented by Coq's `bool` datatype.

Other differences include the key being stored explicitly (instead of being part of a context) and `Z` being used to represent mathematical values. The abstract state is defined as an **`Inductive`** data type instead of a tuple merely to influence the names that Coq assigns to the individual components in proofs.

The other representation will be called the *concrete* DRBG state. The concrete DRBG state will model the exact bytes in the C struct:

```
Definition hmac256drbgstate :=
  (mdstate * (list val * (val * (val * (val * val))))).
```

The concrete state is required to be a tuple of this type (to match the fields of the C struct) by the VST framework; I just gave it a convenient name. Here, `mdstate` is the byte value of the struct representing the HMAC context. `val` is used to represent the bytes representation of the fields in the C struct.

I defined the equivalence between an abstract state and a concrete state:

```
Definition hmac256drbg_relate (a: hmac256drbgabs) (r:
    hmac256drbgstate) : mpred :=
  match a with HMAC256DRBGabs key V reseed_counter entropy_len
      prediction_resistance reseed_interval ⇒
    match r with (md_ctx', (V', (reseed_counter', (entropy_len',
        (prediction_resistance', reseed_interval'))))) ⇒
      md_full key md_ctx'
        && !! (
          map Vint (map Int.repr V) = V'
          ∧ Vint (Int.repr reseed_counter) = reseed_counter'
          ∧ Vint (Int.repr entropy_len) = entropy_len'
          ∧ Vint (Int.repr reseed_interval) = reseed_interval'
          ∧ Val.of_bool prediction_resistance = prediction_resistance'
          )
    end
  end.
```

Notice that I require the HMAC context to be in the `FULL` state, which means that the HMAC context has previously been initialized with some key, and it has to be reset before digesting more data, necessarily throwing away any data that may have been digested before.

Lastly, I defined the conversions between the abstract state and the functional state:

```
Definition hmac256drbgabs_of_state_handle (a: DRBG_state_handle)
    entropy_len reseed_interval: hmac256drbgabs :=
  match a with ((V, key, reseed_counter),_, prediction_resistance) ⇒
    HMAC256DRBGabs key V reseed_counter entropy_len
        prediction_resistance reseed_interval
  end.


Definition hmac256drbgabs_to_state_handle (a: hmac256drbgabs):
    DRBG_state_handle :=
  match a with HMAC256DRBGabs key V reseed_counter entropy_len
      prediction_resistance reseed_interval ⇒
    ((V, key, reseed_counter), 256 (* security strength, not used *),
        prediction_resistance)
  end.
```

4.3. A Description of an API Specification

The functional specification is a mathematical description of a function; it describes, logically, what the output should be given the input. C programs, however, are all about local variables and memory. In order to specify the behavior of a C program, there needs to be a way to describe how the local variables and memory are changed by the program. This is where the API specification comes in.

(As a side note, everything up to this point is in vanilla Coq. The VST framework only comes in when writing the API specification and verifying the C program.)

To write an API specification, the VST framework uses separation logic, a variant of Hoare logic that is better at dealing with pointers. An API specification is of the form:

```
DECLARE function_name
  WITH variables_list
  PRE [ parameters_list ]
     PROP ()
     LOCAL ()
     SEP ()
 POST [ return_type ]
     PROP ()
     LOCAL ()
     SEP ()
```

This is a custom notation defined by the VST framework. Roughly, the API specification states that for all `variables_list`, if the precondition is satisfied, then the function pointed to by `function_name` is allowed to run. If the function terminates, then the postcondition will be satisfied.

Both the pre- and post-conditions are of the form **PROP**`()`**LOCAL**`()`**SEP**`()`. This is separation logic. As an example, take the following:

```
...
  WITH length: Z, buffer: val
  PRE [ _buffer_length OF tuint; _buffer OF tptr tvoid ]
     PROP (0 <= length <= Int.max_unsigned)
     LOCAL (temp _buffer_length (Vint (Int.repr length)); temp _buffer
        buffer)
     SEP (memory_block Tsh length buffer)
...
```

Here, `length` and `buffer` are Coq (logical) variables, while `_buffer_length` and `_buffer` are the names of local variables (function parameters) of the C program. C program identifiers are represented in VST as unique positive integers.

The **PROP** part contains logical statements, in this case saying that the integer `length` is in range for a C unsigned integer. The **LOCAL** part relates identifiers to logical variables. In this case, the **LOCAL** part states that the function parameter `_buffer_length` contains a C unsigned integer equal to the integer `length`. It also relates the function parameter `_buffer` to some bytes `buffer`. The **SEP** part deals with memory, stating that at the area of memory pointed to by `buffer`, `length` bytes have been allocated. Note that an invariant enforced by separation logic is that different clauses in **SEP** always refers to disjoint parts in memory.

As an example, here is the API specification of the HMAC DRBG `update` function, with the less important parts truncated. For a full description of this API specification along with the API specification of other HMAC DRBG functions, please see Appendix B.

```
Definition hmac_drbg_update_spec :=
  DECLARE _mbedtls_hmac_drbg_update
   WITH contents: list Z,
        additional: val, add_len: Z,
        ctx: val, initial_state: hmac256drbgstate,
        initial_state_abs: hmac256drbgabs,
        ...
    PRE [ _ctx OF (tptr t_struct_hmac256drbg_context_st), _additional
        OF (tptr tuchar), _add_len OF tuint ]
       PROP (...)
       LOCAL (temp _ctx ctx; ...)
       SEP (
         (data_at Tsh (tarray tuchar add_len) (map Vint (map Int.repr
            contents)) additional);
         (data_at Tsh t_struct_hmac256drbg_context_st initial_state ctx);
```

```
      ...
       )
  POST [ tvoid ]
    PROP ()
    LOCAL ()
    SEP (
      (hmac256drbgabs_common_mpreds (hmac256drbgabs_hmac_drbg_update
         initial_state_abs contents) initial_state ctx info_contents);
      ...
    ).
```

The DRBG state is pointed to by the pointer value `ctx`. By running this function, assuming the preconditions hold, the state is updated by replacing the `key` and `value` components with the new values obtained by calling `update`, which is what the definition `hmac256drbgabs_hmac_drbg_update` does.

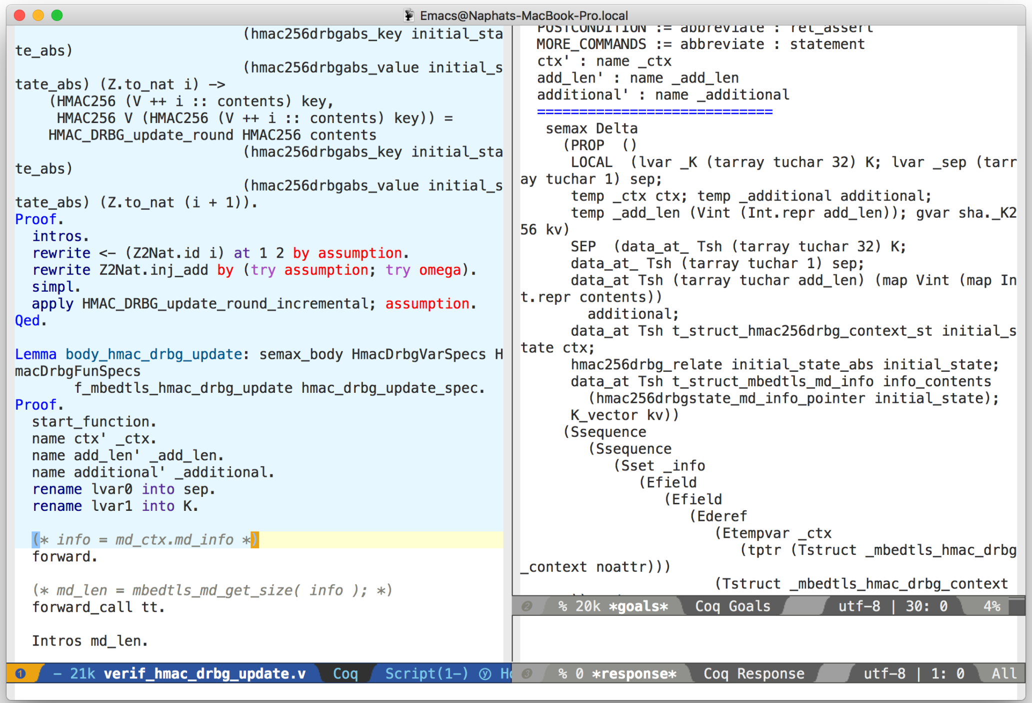
4.4. Proof Process

In this section, I will explain what a typical proof process looks like. I will not be showing complete proofs that the **mbedTLS** functions conform to the API specifications, as such proofs are long and tedious. Fortunately, the proofs are machine-checked in Coq, and so can be trusted to be correct.

Figure 1 shows a part of the `update` function proof. The left pane shows the actual proof, while the right pane shows the current proof goal.

On the left pane, everything highlighted in blue is the part of the proof that has been accepted by the Coq compiler as correct. Everything in white has not been accepted, and so can still be modified. Completing the proof means compiling the entire file, or turning the entire file blue.

In the current proof goal is the proof environment, of the same **`PROP`**`()`**`LOCAL`**`()`**`SEP`**`()` form as seen previously in §4.3. The proof environment contains propositional facts, facts about local variables, and facts about memory.



Emacs@Naphats-MacBook-Pro.local

```
                                   (hmac256drbgabs_key initial_sta
te_abs)
                                   (hmac256drbgabs_value initial_s
tate_abs) (Z.to_nat i) ->
    (HMAC256 (V ++ i :: contents) key,
     HMAC256 V (HMAC256 (V ++ i :: contents) key)) =
    HMAC_DRBG_update_round HMAC256 contents
                                   (hmac256drbgabs_key initial_sta
te_abs)
                                   (hmac256drbgabs_value initial_s
tate_abs) (Z.to_nat (i + 1)).
Proof.
  intros.
  rewrite <- (Z2Nat.id i) at 1 2 by assumption.
  rewrite Z2Nat.inj_add by (try assumption; try omega).
  simpl.
  apply HMAC_DRBG_update_round_incremental; assumption.
Qed.

Lemma body_hmac_drbg_update: semax_body HmacDrbgVarSpecs H
macDrbgFunSpecs
       f_mbedtls_hmac_drbg_update hmac_drbg_update_spec.
Proof.
  start_function.
  name ctx' _ctx.
  name add_len' _add_len.
  name additional' _additional.
  rename lvar0 into sep.
  rename lvar1 into K.

  (* info = md_ctx.md_info *)
  forward.

  (* md_len = mbedtls_md_get_size( info ); *)
  forward_call tt.

  Intros md_len.
```

```
- 21k verif_hmac_drbg_update.v   Coq   Script(1-)   Ho
```

```
  POSTCONDITION := abbreviate : ret_assert
  MORE_COMMANDS := abbreviate : statement
  ctx' : name _ctx
  add_len' : name _add_len
  additional' : name _additional
  ============================
   semax Delta
     (PROP  ()
      LOCAL  (lvar _K (tarray tuchar 32) K; lvar _sep (tarr
ay tuchar 1) sep;
      temp _ctx ctx; temp _additional additional;
      temp _add_len (Vint (Int.repr add_len)); gvar sha._K2
56 kv)
      SEP  (data_at_ Tsh (tarray tuchar 32) K;
      data_at_ Tsh (tarray tuchar 1) sep;
      data_at Tsh (tarray tuchar add_len) (map Vint (map In
t.repr contents))
        additional;
      data_at Tsh t_struct_hmac256drbg_context_st initial_s
tate ctx;
      hmac256drbg_relate initial_state_abs initial_state;
      data_at Tsh t_struct_mbedtls_md_info info_contents
        (hmac256drbgstate_md_info_pointer initial_state);
      K_vector kv))
     (Ssequence
        (Ssequence
           (Sset _info
              (Efield
                 (Efield
                    (Ederef
                       (Etempvar _ctx
                          (tptr (Tstruct _mbedtls_hmac_drbg
_context noattr)))
                       (Tstruct _mbedtls_hmac_drbg_context
```

```
% 20k *goals*   Coq Goals   utf-8 | 30: 0   4%
```

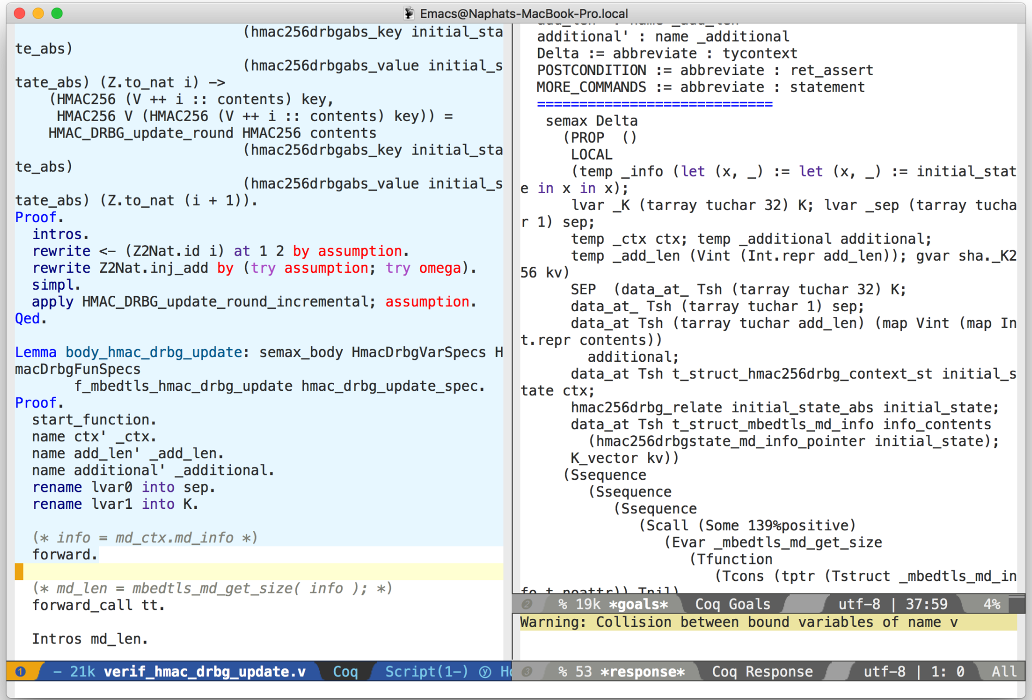
```
% 0 *response*   Coq Response   utf-8 | 1: 0   All
```

Figure 1: The `update` Function Proof

The current proof goal also contains the abstract syntax tree (AST) of the C code. Each statement in the AST modifies the proof environment, and each step in the proof (left pane) is an instruction to Coq on how to step through the AST. For simple statements, such as an assignment statement, the proof automation system can figure out how the proof environment will be affected by stepping through the statement. Other statements require the proof to explicitly state how the proof environment will be affected. These statements include the **`if`** statement or different kinds of loops.

Figure 1 shows the case where the next statement in the AST is an assignment statement. This is the simple case, and so the proof command `forward` can automatically figure out how the environment will be modified. Figure 2 shows the same proof after the `forward` command has been accepted by the Coq compiler. Notice the proof environment has changed in the **`LOCAL`** part, and the AST no longer shows the assignment statement as the next statement.

The proof environment is initialized to the function preconditions. A complete proof shows that, after the entire AST has been stepped through, the modified environment implies the function



```
                                   (hmac256drbgabs_key initial_sta
te_abs)
                                   (hmac256drbgabs_value initial_s
tate_abs) (Z.to_nat i) ->
    (HMAC256 (V ++ i :: contents) key,
     HMAC256 V (HMAC256 (V ++ i :: contents) key)) =
    HMAC_DRBG_update_round HMAC256 contents
                                   (hmac256drbgabs_key initial_sta
te_abs)
                                   (hmac256drbgabs_value initial_s
tate_abs) (Z.to_nat (i + 1)).
Proof.
  intros.
  rewrite <- (Z2Nat.id i) at 1 2 by assumption.
  rewrite Z2Nat.inj_add by (try assumption; try omega).
  simpl.
  apply HMAC_DRBG_update_round_incremental; assumption.
Qed.

Lemma body_hmac_drbg_update: semax_body HmacDrbgVarSpecs H
macDrbgFunSpecs
       f_mbedtls_hmac_drbg_update hmac_drbg_update_spec.
Proof.
  start_function.
  name ctx' _ctx.
  name add_len' _add_len.
  name additional' _additional.
  rename lvar0 into sep.
  rename lvar1 into K.

  (* info = md_ctx.md_info *)
  forward.

  (* md_len = mbedtls_md_get_size( info ); *)
  forward_call tt.

  Intros md_len.
```

```
  additional' : name _additional
  Delta := abbreviate : tycontext
  POSTCONDITION := abbreviate : ret_assert
  MORE_COMMANDS := abbreviate : statement
  ============================
   semax Delta
     (PROP  ()
      LOCAL
      (temp _info (let (x, _) := let (x, _) := initial_stat
e in x in x);
      lvar _K (tarray tuchar 32) K; lvar _sep (tarray tucha
r 1) sep;
      temp _ctx ctx; temp _additional additional;
      temp _add_len (Vint (Int.repr add_len)); gvar sha._K2
56 kv)
      SEP  (data_at_ Tsh (tarray tuchar 32) K;
      data_at_ Tsh (tarray tuchar 1) sep;
      data_at Tsh (tarray tuchar add_len) (map Vint (map In
t.repr contents))
        additional;
      data_at Tsh t_struct_hmac256drbg_context_st initial_s
tate ctx;
      hmac256drbg_relate initial_state_abs initial_state;
      data_at Tsh t_struct_mbedtls_md_info info_contents
        (hmac256drbgstate_md_info_pointer initial_state);
      K_vector kv))
     (Ssequence
        (Ssequence
           (Ssequence
              (Scall (Some 139%positive)
                 (Evar _mbedtls_md_get_size
                    (Tfunction
                       (Tcons (tptr (Tstruct _mbedtls_md_in
```

```
Warning: Collision between bound variables of name v
```

Figure 2: After Compiling the `forward` Command

postconditions. Figure 3 shows when the entire proof has been compiled. Stepping through the **`Qed`** command will complete the proof.

4.5. Verification Statistics

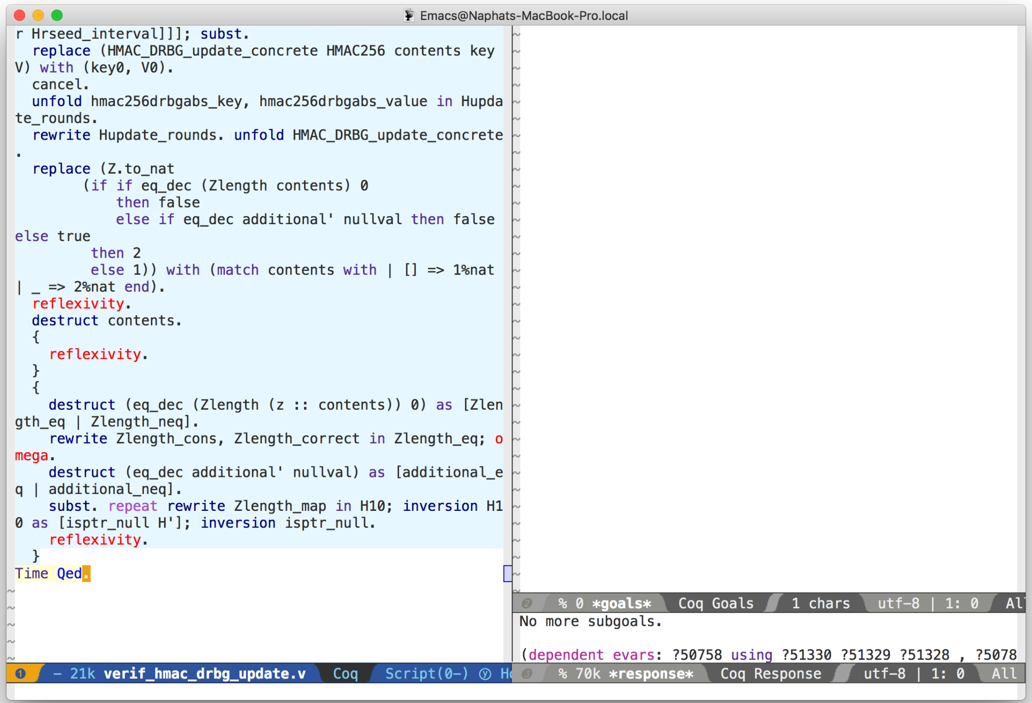
I verified the **mbedTLS** implementations of `update`, `reseed`, and `generate` correct with respect to their API specifications. In this section, I will provide some statistics about the verifications.

Function	C (lines)	API Spec. (lines)	Proof (lines)	Proof (compilation time)
`update`	32	32	552	28 minutes
`reseed`	41	38	710	18 minutes
`generate`	69	48	2456	35 minutes

Table 1: Verification Statistics

Line counts here include empty lines as well as comments. The compilation time was measured on a Macbook Pro with Intel Core i7, 2.5 GHz, 16 gigabytes of memory.

The actual verifications are significantly longer than the C counterpart. Fortunately, they are



```
r Hrseed_interval]]]; subst.
  replace (HMAC_DRBG_update_concrete HMAC256 contents key
V) with (key0, V0).
  cancel.
  unfold hmac256drbgabs_key, hmac256drbgabs_value in Hupda
te_rounds.
  rewrite Hupdate_rounds. unfold HMAC_DRBG_update_concrete
.
  replace (Z.to_nat
         (if if eq_dec (Zlength contents) 0
             then false
             else if eq_dec additional' nullval then false
else true
          then 2
          else 1)) with (match contents with | [] => 1%nat
| _ => 2%nat end).
  reflexivity.
  destruct contents.
  {
    reflexivity.
  }
  {
    destruct (eq_dec (Zlength (z :: contents)) 0) as [Zlen
gth_eq | Zlength_neq].
    rewrite Zlength_cons, Zlength_correct in Zlength_eq; o
mega.
    destruct (eq_dec additional' nullval) as [additional_e
q | additional_neq].
    subst. repeat rewrite Zlength_map in H10; inversion H1
0 as [isptr_null H']; inversion isptr_null.
    reflexivity.
  }
Time Qed.
```

```
% 0 *goals*   Coq Goals   1 chars   utf-8 | 1: 0   All
No more subgoals.

(dependent evars: ?50758 using ?51330 ?51329 ?51328 , ?5078
```

Figure 3: At the End of the `update` Proof

machine-checked, and as long as they compile successfully, it is unlikely that clients of my project would need to take a look at the actual verifications. Instead, the API specifications will be of more interest to potential clients.

5. Difficulties

One of the goals of my project is to serve as a bleeding-edge adopter of the VST framework and the HMAC API specification. As is the nature of bleeding-edge adoption, I encountered many difficulties, such as proof tactics failing silently and Verifiable C's theory of pointer comparisons being too weak. These difficulties were fixed as they got reported back to the developers of VST. In this section, I highlight one particular difficulty with the HMAC API specification.

5.1. HMAC API Specification

The C (**OpenSSL**) implementation of SHA-256 HMAC defines the HMAC context as a struct with three SHA-256 contexts. The original API specification of HMAC [3] uses this same representation as the abstract state of HMAC.

```
Inductive hmacabs :=
  HMACabs: ∀(ctx iSha oSha: s256abs) hmacabs.
```

This representation works in the sense that the verification of the **OpenSSL** functions went through. Unfortunately, for clients of this API specification, this representation is often not good enough. The biggest issue is that this representation is too low-level; it does not keep track of the key. Upon being notified of this issue, the developers of the HMAC API specification were quickly able to provide a more abstract representation of the HMAC state, leading to the state machine representation shown earlier in §4.1. The core of the verification code did not need to be changed, with some book-keeping added at the ends of translate between the new, more abstract representation to the more low-level implementation. While this issue was easily fixed, it would have been very hard to discover this issue without actually being clients of the API specification. This highlights the importance of having being the first client of the HMAC API specification.

6. Conclusion

I verified the **mbedTLS** implementations of HMAC DRBG `update`, `reseed`, and `generate`. I wrote functional specifications of these functions, describing mathematically what the functions do as functional programs. Using the functional specifications, I wrote the API specifications to describe how C implementations of these functions are required to interact with local variables and memory. I then verified the **mbedTLS** implementations correct with respect to the API specifications. While the verifications themselves were long and unwieldy, they are machine-checked and so are not part of the code that needs to be trusted. While the functional specifications must still be trusted, they will not have to be once Katherine Ye's parallel project on proving cryptographic properties of the functional specifications complete in the spring of 2016. Along the way, I identified some issues with the VST framework and the HMAC API specifications, and those issues are now fixed by the relevant parties. My project, along with the verification of the HMAC DRBG `instantiate` function, will serve as one of the building blocks that future verification tasks can use.

References

[1] A. W. Appel, “Verification of a cryptographic primitive: SHA-256,” *ACM Transactions on Programming Languages and Systems*, vol. 37, no. 2, pp. 7:1–7:31, Apr. 2015.

[2] E. B. Barker and J. M. Kelsey, “Sp 800-90a. recommendation for random number generation using deterministic random bit generators,” Gaithersburg, MD, United States, Tech. Rep., 2012.

[3] L. Beringer *et al.*, “Verified correctness and security of openSSL HMAC,” in *USENIX Security*, J. Jung and T. Holz, Eds. USENIX Association, 2015, pp. 207–221.

[4] F. Dörre and V. Klebanov, “Pseudo-random number generator verification: A case study,” in *Proceedings, Verified Software: Theories, Tools, and Experiments (VSTTE)*, A. Gurfinkel and S. A. Seshia, Eds., 2015, to appear.

Appendices

A. Bridging OpenSSL's HMAC with mbedTLS's Message Digest Interface

As previously mentioned in §4.1, previous verification efforts on HMAC and SHA are done on **OpenSSL**. Therefore, the **mbedTLS** DRBG implementation must somehow call into the **OpenSSL** HMAC implementation, so that the API specifications for HMAC can be used. To do this, I wrote a thin wrapper around the **mbedTLS** message digest interface. In this section, I show exactly how I did that.

The **mbedTLS** message digest interface declares the following functions:

```
int mbedtls_md_setup( mbedtls_md_context_t *ctx, const
    mbedtls_md_info_t *md_info, int hmac );

void mbedtls_md_free( mbedtls_md_context_t *ctx );

unsigned char mbedtls_md_get_size( const mbedtls_md_info_t *md_info );

int mbedtls_md_hmac_starts( mbedtls_md_context_t *ctx, const unsigned
    char *key, size_t keylen );

int mbedtls_md_hmac_update( mbedtls_md_context_t *ctx, const unsigned
    char *input, size_t ilen );

int mbedtls_md_hmac_finish( mbedtls_md_context_t *ctx, unsigned char
    *output);

int mbedtls_md_hmac_reset( mbedtls_md_context_t *ctx );
```

A.1. The HMAC Context

In order for clients of the **mbedTLS** message digest interface to use **OpenSSL** HMAC implementation, the clients must somehow have access to **OpenSSL** HMAC context. The clients already have access to the **mbedTLS** message digest context, which is defined as the following struct:

```
typedef struct mbedtls_md_context_t {
    const mbedtls_md_info_t *md_info;
    void *md_ctx;
    void *hmac_ctx;
} mbedtls_md_context_t;
```

The `md_info` field can mostly be ignored, as it is used to differentiate between the different types of message digest, and in this case, we are specializing the message digest to use HMAC and SHA-256. In order for clients of the message digest interface to have access to the **OpenSSL** HMAC context, I chose to have the `hmac_ctx` pointer point to the **OpenSSL** HMAC context. This initialization happens in the `mbedtls_md_setup` function:

```
int mbedtls_md_setup( mbedtls_md_context_t *ctx, const
    mbedtls_md_info_t *md_info, int hmac ) {
    HMAC_CTX * sha_ctx = (HMAC_CTX *) malloc(sizeof(HMAC_CTX));
    if (sha_ctx == NULL) {
        return MBEDTLS_ERR_MD_ALLOC_FAILED;
    }
    ctx->hmac_ctx = sha_ctx;
    return 0;
}
```

The corresponding `mbedtls_md_free` function is done in an obvious way:

```
void mbedtls_md_free( mbedtls_md_context_t *ctx ) {
    HMAC_CTX * hmac_ctx = ctx->hmac_ctx;
    free(hmac_ctx);
}
```

A.2. Other Functions

The function `mbedtls_md_get_size` can be specialized to ignore `md_info` and always return the length of the output of SHA-256 HMAC, which is 32 bytes.

```
unsigned char mbedtls_md_get_size( const mbedtls_md_info_t *md_info ) {
    return SHA256_DIGEST_LENGTH;
}
```

The functions `mbedtls_md_hmac_starts`, `mbedtls_md_hmac_update`, and `mbedtls_md_hmac_finish` have counterparts in the **OpenSSL** library, and so can just call through to them.

```
int mbedtls_md_hmac_starts( mbedtls_md_context_t *ctx, const unsigned
    char *key, size_t keylen ) {
    HMAC_CTX * hmac_ctx = ctx->hmac_ctx;
    HMAC_Init(hmac_ctx, (unsigned char*) key, keylen);
    return 0;
}


int mbedtls_md_hmac_update( mbedtls_md_context_t *ctx, const unsigned
    char *input, size_t ilen ) {
    HMAC_CTX * hmac_ctx = ctx->hmac_ctx;
    HMAC_Update(hmac_ctx, (const void *) input, ilen);
    return 0;
}
```

```
int mbedtls_md_hmac_finish( mbedtls_md_context_t *ctx, unsigned char
    *output) {
    HMAC_CTX * hmac_ctx = ctx->hmac_ctx;
    HMAC_Final(hmac_ctx, output);
    return 0;
}
```

OpenSSL does not have an explicit `reset` function. Recall that the `reset` function takes a HMAC context `FULL key` and takes it to `REP key []`. That is, it makes the HMAC context ready for digesting data with the pre-existing key. In OpenSSL, this is done by passing `NULL` as the key to `HMAC_Init`.

```
int mbedtls_md_hmac_reset( mbedtls_md_context_t *ctx ) {
    HMAC_CTX * hmac_ctx = ctx->hmac_ctx;
    HMAC_Init(hmac_ctx, NULL, SHA256_DIGEST_LENGTH);
    return 0;
}
```

B. DRBG API Specifications

In this section, I present a detailed description of the API specifications for the following functions: `update`, `reseed`, and `generate`.

B.1. The Update Function

The API specification is as follows:

```
Definition hmac_drbg_update_spec :=
  DECLARE _mbedtls_hmac_drbg_update
   WITH contents: list Z,
        additional: val, add_len: Z,
        ctx: val, initial_state: hmac256drbgstate,
   initial_state_abs: hmac256drbgabs,
   kv: val, info_contents: md_info_state
PRE [ _ctx OF (tptr t_struct_hmac256drbg_context_st), _additional
   OF (tptr tuchar), _add_len OF tuint ]
  PROP (
    0 <= add_len <= Int.max_unsigned;
    Zlength (hmac256drbgabs_value initial_state_abs) = Z.of_nat
       SHA256.DigestLength;
    add_len = Zlength contents;
    Forall isbyteZ (hmac256drbgabs_value initial_state_abs);
    Forall isbyteZ contents
  )
  LOCAL (temp _ctx ctx; temp _additional additional; temp _add_len
     (Vint (Int.repr add_len)); gvar sha._K256 kv)
  SEP (
    (data_at Tsh (tarray tuchar add_len) (map Vint (map Int.repr
       contents)) additional);
    (data_at Tsh t_struct_hmac256drbg_context_st initial_state ctx);
    (hmac256drbg_relate initial_state_abs initial_state);
    (data_at Tsh t_struct_mbedtls_md_info info_contents
       (hmac256drbgstate_md_info_pointer initial_state));
    (K_vector kv)
     )
POST [ tvoid ]
  PROP ()
  LOCAL ()
  SEP (
    (hmac256drbgabs_common_mpreds (hmac256drbgabs_hmac_drbg_update
       initial_state_abs contents) initial_state ctx info_contents);
    (data_at Tsh (tarray tuchar add_len) (map Vint (map Int.repr
       contents)) additional);
    (K_vector kv)
  ).
```

The first **WITH** variable is `contents`, which is mathematical representation of the sequence of bytes to be consumed by the update function (`provided_data` in the functional specification). The next two **WITH** values are the pointer value, `additional`, at which `contents` is located in memory, and the length of `contents`, `add_len`. In the **PROP** part of the specification, `add_len` is restricted to be a valid unsigned integer, and the length of `contents` is enforced to be `add_len`. The mathematical integers in `contents` are also enforced to be valid bytes. In the **SEP** part, `additional` is said to point to `contents`. In the postcondition, the same **SEP** clause appears, showing that the update function does not change the memory pointed to by `additional`.

The next **WITH** value is the pointer value, `ctx`, pointing to the DRBG struct. The next two **WITH** values are the abstract and concrete DRBG states, respectively. The last **WITH** value is the content of the struct denoting what kind of HMAC is being used, `info_contents`. In the **PROP** clause, the sequence `V` part of the abstract state is enforced to be the size of the output of the HMAC function and enforced to contain valid bytes. This is an invariant that the update function and all public functions enforce, and so will be seen throughout all API specifications. In the **SEP** clause, `ctx` is said to point to an area in memory that contains the concrete state, and the concrete state is related to the abstract state by the equivalence definition. The info pointer from the concrete state also points to a valid address containing `info_contents`. In the post condition, there is a new abstract state calculated by `hmac256drbgabs_hmac_drbg_update`, which is just a wrapper around the functional specification with the HMAC256 function used. In the **SEP** part, there exists a corresponding concrete state pointed to by `ctx` that is equivalent to the new abstract state. This new concrete state's info pointer also points to the same valid memory location containing info_contents. This is what the definition hmac256drbgabs_common_mpreds encapsulates.

The **WITH** value kv represents the pointer value to the SHA K vector, which is a global constant array in the SHA-256 implementation. For more information, see the verification of SHA-256 [1].

B.2. The Reseed Function

The API specification for the reseed function is as follows:

```
Definition hmac_drbg_reseed_spec :=
  DECLARE _mbedtls_hmac_drbg_reseed
   WITH contents: list Z,
        additional: val, add_len: Z,
        ctx: val, initial_state: hmac256drbgstate,
        initial_state_abs: hmac256drbgabs,
        kv: val, info_contents: md_info_state,
        s: ENTROPY.stream
    PRE [ _ctx OF (tptr t_struct_hmac256drbg_context_st), _additional
        OF (tptr tuchar), _len OF tuint ]
       PROP (
        0 <= add_len <= Int.max_unsigned;
        Zlength (hmac256drbgabs_value initial_state_abs) = Z.of_nat
            SHA256.DigestLength;
        add_len = Zlength contents;
        hmac256drbgabs_entropy_len initial_state_abs = 32;
        Forall isbyteZ (hmac256drbgabs_value initial_state_abs);
        Forall isbyteZ contents
       )
       LOCAL (temp _ctx ctx; temp _additional additional; temp _len
          (Vint (Int.repr add_len)); gvar sha._K256 kv)
    SEP (
      (data_at Tsh (tarray tuchar add_len) (map Vint (map Int.repr
         contents)) additional);
      (data_at Tsh t_struct_hmac256drbg_context_st initial_state ctx);
      (hmac256drbg_relate initial_state_abs initial_state);
      (data_at Tsh t_struct_mbedtls_md_info info_contents
         (hmac256drbgstate_md_info_pointer initial_state));
      (Stream s);
      (K_vector kv)
       )
POST [ tint ]
    EX ret_value:_,
    PROP (
       return_value_relate_result
          (mbedtls_HMAC256_DRBG_reseed_function s initial_state_abs
          contents) ret_value
      )
    LOCAL (temp ret_temp ret_value)**
    SEP (
      (hmac256drbgabs_common_mpreds (hmac256drbgabs_reseed
         initial_state_abs s contents) initial_state ctx
         info_contents);
      (data_at Tsh (tarray tuchar add_len) (map Vint (map Int.repr
         contents)) additional);
      (Stream (get_stream_result
         (mbedtls_HMAC256_DRBG_reseed_function s initial_state_abs
         contents)));
```

```
          (K_vector kv)
        ).
```

Notice a new notation here: **EX** `ret_value:_`. This can be read literally as "there exists a variable `ret_value` that satisfies the postcondition". This existential variable captures the return value of the function, and its value is enforced in the **PROP** part of the postcondition. The return value is enforced to be 0 if the reseed succeeds, a specific nonzero value if the reseed fails because of a failure to get entropy, and any nonzero value if the reseed fails for any other reason.

`mbedtls_HMAC256_DRBG_reseed_function` is just a wrapper around the functional specification of the reseed function, with the reseed algorithm and other dependencies filled in. `hmac256drbgabs_reseed` returns the resulting abstract state from the reseed function if the reseed succeeds, or the initial abstract state otherwise.

Much like the `update` function, the first three **WITH** values are used for the additional data parameter. `additional` is a pointer value that points to `contents`, of length `add_len`, and the function leaves this relation untouched.

Again, much like the `update` function, the next three **WITH** values are the pointer value to the DRBG state, the abstract DRBG state, and the concrete DRBG state. The only new property is in the **PROP** part of the precondition, requiring the `entropy_len` component to be 32. In **NIST SP 800-90A** and therefore in the functional specification, this is a global, fixed constant, but **mbedTLS** varies this value based on the type of HMAC used and fixes the value upon instantiation. With HMAC256, this is fixed to be 32.

Unlike the `update` function, the `reseed` function needs to reason about entropy; that is the whole point of reseeding! The **WITH** value `s` represents the entropy stream to be used, and a new one is passed back out in the **SEP** part of the postcondition.

B.3. The Generate Function

The API specification for the `generate` function is as follows:

```
Definition hmac_drbg_generate_spec :=
DECLARE _mbedtls_hmac_drbg_random_with_add
 WITH contents: list Z,
    additional: val, add_len: Z,
    output: val, out_len: Z,
    ctx: val, initial_state: hmac256drbgstate,
    initial_state_abs: hmac256drbgabs,
    kv: val, info_contents: md_info_state,
    s: ENTROPY.stream
  PRE [ _p_rng OF (tptr tvoid), _output OF (tptr tuchar), _out_len OF
    tuint, _additional OF (tptr tuchar), _add_len OF tuint ]
   PROP (
    0 <= add_len <= Int.max_unsigned;
    0 <= out_len <= Int.max_unsigned;
    Zlength (hmac256drbgabs_value initial_state_abs) = Z.of_nat
       SHA256.DigestLength;
    add_len = Zlength contents;
    hmac256drbgabs_entropy_len initial_state_abs = 32;
    hmac256drbgabs_reseed_interval initial_state_abs = 10000;
    0 <= hmac256drbgabs_reseed_counter initial_state_abs <=
       Int.max_signed;
    Forall isbyteZ (hmac256drbgabs_value initial_state_abs);
    Forall isbyteZ contents
   )
   LOCAL (temp _p_rng ctx; temp _output output; temp _out_len (Vint
      (Int.repr out_len)); temp _additional additional; temp _
      add_len (Vint (Int.repr add_len)); gvar sha._K256 kv)
   SEP (
    (data_at_ Tsh (tarray tuchar out_len) output);
    (data_at Tsh (tarray tuchar add_len) (map Vint (map Int.repr
       contents)) additional);
    (data_at Tsh t_struct_hmac256drbg_context_st initial_state ctx);
    (hmac256drbg_relate initial_state_abs initial_state);
    (data_at Tsh t_struct_mbedtls_md_info info_contents
       (hmac256drbgstate_md_info_pointer initial_state));
    (Stream s);
    (K_vector kv)
     )
POST [ tint ]
  EX ret_value:_,
  PROP (
     return_value_relate_result
        (mbedtls_HMAC256_DRBG_generate_function s
        initial_state_abs out_len contents) ret_value
   )
  LOCAL (temp ret_temp ret_value)
  SEP (
    (match mbedtls_HMAC256_DRBG_generate_function s
       initial_state_abs out_len contents with
      | ENTROPY.error _ => (data_at_ Tsh (tarray tuchar out_len)
         output)
      | ENTROPY.success (bytes, _) => (data_at Tsh (tarray tuchar
         out_len) (map Vint (map Int.repr bytes)) output)
     end
    );
    (hmac256drbgabs_common_mpreds (hmac256drbgabs_generate
       initial_state_abs s out_len contents) initial_state ctx
       info_contents);
    (data_at Tsh (tarray tuchar add_len) (map Vint (map Int.repr
       contents)) additional);
    (Stream (get_stream_result
       (mbedtls_HMAC256_DRBG_generate_function s initial_state_abs
       out_len contents)));
    (K_vector kv)
  ).
```

The main difference between this specification and the API specification of `reseed` is that the generate function takes as input a pointer (`output`) to a buffer of size `out_len`, which is filled with the resulting stream of bytes on success and left alone on failure.